

Encoding Object-oriented Datatypes in HOL: Extensible Records Revisited

The HOL-OCL Experience

Achim D. Brucker
achim@brucker.ch http://www.brucker.ch/

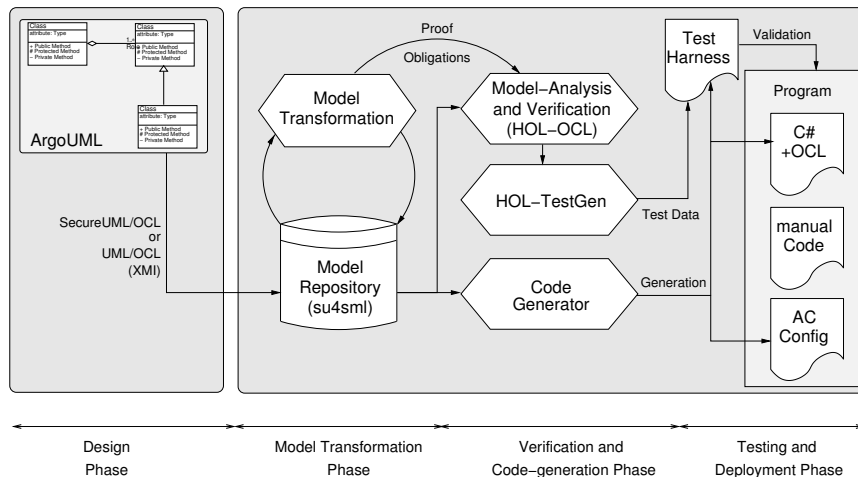
Isabelle Developers Workshop (IDW 2010)
Cambridge, UK, 17th June 2010

Outline

- 1 Introduction
- 2 An extensible Encoding of Object-oriented Data Models in HOL
- 3 HOL-OCL
- 4 Outlook and Conclusion

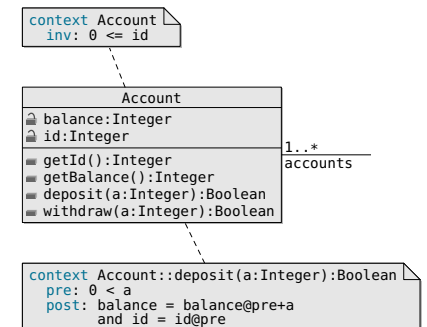
Our Vision:

Tool Supported Formal Methods for (Model-driven) Software Development



UML/OCL in a Nutshell

- UML
 - Visual modeling language
 - Object-oriented development
 - Industrial tool support
 - OMG standard
 - Many diagram types, e. g.,
 - activity diagrams
 - class diagrams
 - ...
- OCL
 - Textual extension of the UML
 - Allows for annotating UML diagrams
 - In the context of class-diagrams:
 - invariants
 - preconditions
 - postconditions



Developing Formals Tools for UML/OCL?

Turning UML/OCL into a formal method

- 1 A formal semantics of **object-oriented data models** (UML)
 - typed path expressions
 - inheritance
 - ...
- 2 A formal semantics of **object-oriented constraints** (OCL)
 - a logic reasoning over path expressions
 - large libraries
 - three-valued logic
 - ...
- 3 And of course, we want a tool (**HOL-OCL**)
 - a formal, machine-checked semantics for OO specifications,
 - an interactive proof environment for OO specifications.

Challenges (for a shallow embedding)

■ Challenge 1:

Can we find a injective, type preserving mapping of an object-oriented language (and datatypes) into HOL

$$e: T \longrightarrow e :: T$$

(including subtyping)?

■ Challenge 2:

Can we support verification in a modular way
(i. e., no replay of proof scripts after extending specifications)?

■ Challenge 3:

Can we ensure consistency of our representation?

Representing Class Types

■ The “extensible records” approach

- We assume a common superclass (0).
- A *tag type* guarantees uniqueness by ($O_{\text{tag}} := \text{classO}$).
- Construct class type as tuple along inheritance hierarchy:

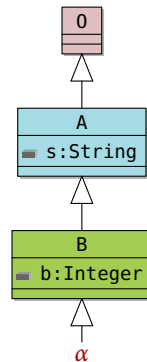
$$\alpha B := (O_{\text{tag}} \times \text{oid}) \times ((A_{\text{tag}} \times \text{String}) \times ((B_{\text{tag}} \times \text{Integer}) \times \alpha))$$

■ Advantages:

- it allows for extending class types (inheritance),
 - subclasses are type instances of superclasses
- ⇒ **it allows for modular proofs**, i. e.,
a statement $\phi(x :: (\alpha B))$ proven for class B is still valid after extending class B.

■ However, it has a major disadvantage:

- modular proofs are only supported for **one** extension per class

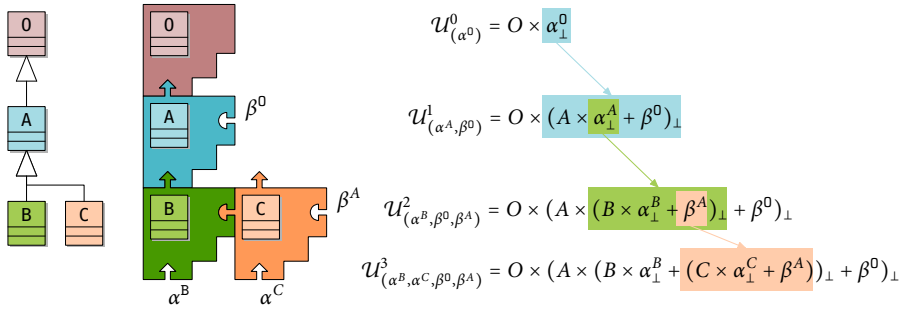


Idea: A General Universe Type

A **universe** type representing all classes of a class model

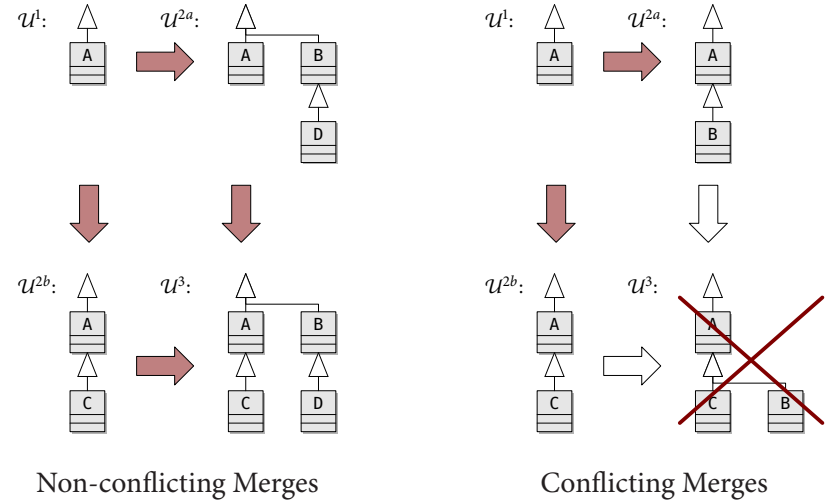
- supports modular proofs with arbitrary extensions
- provides a formalization of a extensible typed object store

An Extensible Object Store



$$U^3_{(\alpha^B, \alpha^C, \beta^0, \beta^A)} < U^2_{(\alpha^B, \beta^0, \beta^A)} < U^1_{(\alpha^A, \beta^0)} < U^0_{(\alpha^0)}$$

Merging Universes



Operations Accessing the Object Store

- injections

$$mk_O o = \text{Inl } o \quad \text{with type } \alpha^O O \rightarrow U^0_{\alpha^O}$$

- projections

$$get_O u = u \quad \text{with type } U^0_{\alpha^O} \rightarrow \alpha^O O$$

- type casts

$$A_{[O]} = get_O \circ mk_A \quad \text{with type } \alpha^A A \rightarrow (A \times \alpha^A_{\perp} + \beta^O) O$$

$$O_{[A]} = get_A \circ mk_O \quad \text{with type } (A \times \alpha^A_{\perp} + \beta^O) O \rightarrow \alpha^A A$$

- ...

All definitions are generated automatically

Does This Really Model Object-orientation?

For each UML model, we have to show several properties:

- subclasses are of the superclasses kind:

$$\frac{\text{isType}_B \text{ self}}{\text{isKind}_A \text{ self}}$$

- “re-casting”:

$$\frac{\text{isType}_B \text{ self}}{\text{self}_{[A][B]} \neq \perp \wedge \text{isType}_B (\text{self}_{[A][B][A]})}$$

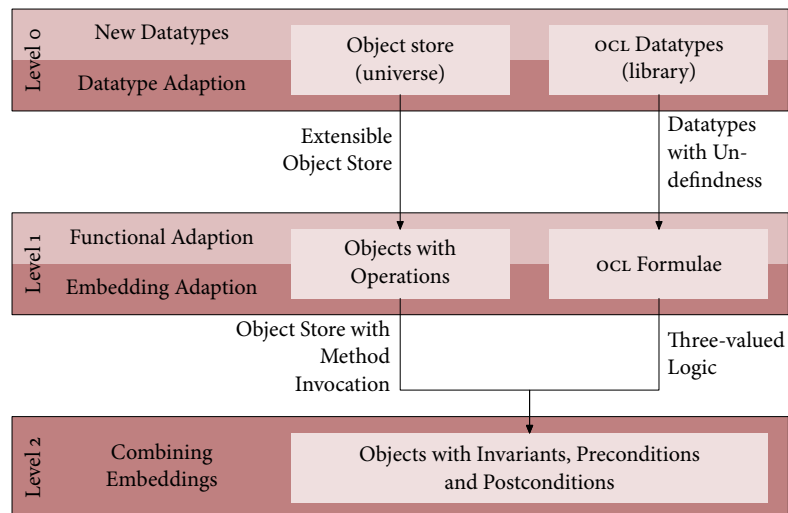
- monotonicity of invariants, ...

All rules are derived automatically

This is only the beginning ...

- **Type-safety** of “object-type accessors” needs further processing.
- **Encoding invariants** requires (co-)inductive definitions.
- **Solution:** encoding based on three levels:
 - 1 weakly typed data types
 - 2 strongly typed data types (and support for operations)
 - 3 constrained data modes

An multi-level object-oriented datatype-package



Encoding Attribute Accessors

Assume a class `Node` with an attribute `next`:

- 1 **Unsafe access** (reference, value, or \perp):

$$self.next^{(0)} \equiv (fst \circ snd \circ snd \circ fst) \uparrow \text{base } self \uparrow$$

- 2 **Type-safe access** (typed object, value, or \perp):

$$self.next^{(1)} \equiv \lambda \sigma. \begin{cases} \text{get}_{Node}^{(0)} u & \text{if } \sigma(self.next^{(0)}) = \perp u \perp \\ \perp & \text{otherwise.} \end{cases}$$

- 3 **Semantically-safe access** (object satisfying invariant, value, or \perp):

$$self.next^{(2)} \equiv \lambda \sigma. \begin{cases} self.next^{(1)} & \text{if } \sigma \models self.next^{(1)} \in \mathfrak{K}_{Node} \\ \perp & \text{otherwise.} \end{cases}$$

where \mathfrak{K}_{Node} is the (co-) inductively defined *characteristic kind* set of class `Node`.

Case Studies (Datatype Package)

- Importing object-oriented models:

	Invoice	eBank	Company	R&L
classes	3	8	7	13
specification (lines)	149	114	210	520
generated theorems	647	1444	1312	2516
time (in seconds)	12	42	49	136

- The core library takes about 20 minutes (1200 seconds) to build
- Extensionality saves about 20 minutes on each import

Challenges (Revisited)

- **Challenge 1:** *Can we find an injective, type preserving mapping of an object-oriented language (and datatypes) into HOL?*
Yes, our encoding is even bijective.
- **Challenge 2:** *Can we support verification in a modular way (i. e., no replay of proof scripts after extending specifications)?*
Yes, a specific form of extensionality can be supported.
- **Challenge 3:** *Can we ensure consistency of our representation?*
Yes, by using a conservative embedding (deriving all rules).

HOL-OCL



- HOL-OCL provides:
 - a formal, machine-checked semantics for OO specifications,
 - an interactive proof environment for OO specifications.
- HOL-OCL is integrated into a toolchain providing:
 - extended well-formedness checking,
 - proof-obligation generation,
 - methodology support for UML/OCL,
 - a transformation framework (including PO generation),
 - code generators,
 - support for SecureUML.
- HOL-OCL is publicly available:
<http://www.brucker.ch/projects/hol-ocl/>.

The HOL-OCL User Interface

```

3 emacs@nakagawa.inf.ethz.ch "1/6"
File Edit Options Buffers Tools Preview LaTeX Command X-Symbol Help
State Context Goal Retract Undo Next Use Goto G.F.D. Find Comment Stop Restart Info Help
\begin{small}
\lstinputlisting[style=ocl]{company.ocl}
\end{small}
\begin{figure}
\centering
\includegraphics[scale=.6]{company}
\caption[A company Class Diagram]{label{fig:company_classdiag}}
\end{figure}
*)
load_xmi "company_ocl.xmi"
thm Company.Person.inv.inv_19_def
lemma "Company.Person.inv self → Company.Person.inv.inv_19 self"
apply(simp add: Company.Person.inv.inv_19_def
         Company.Person.inv.inv_19_def)
apply(auto)
-1- ** Company.thy 802 (45.14) SVN-27978 (Isar_script [PDFLaTeX/F] MMM XS:holocl/s Scripting) ---6:35 2.39
\<<sync>thm Company.Person.inv.inv_19_def: \<<sync>
Person.inv.inv_19 =
λself. ∀ p2 ∈ OclAllInstances
  self • (∀ p1 ∈ OclAllInstances
    self • ((p1 '◁' p2) →
            (Company.Person.lastName p1 '◁' Company.Person.lastName p2)))[]
-1- **response* All (6,101) (response) ---6:35 2.39 Mail
  
```

The HOL-OCL High-level Language

The HOL-OCL proof language is an extension of Isabelle's Isar language:

- importing UML/OCL:

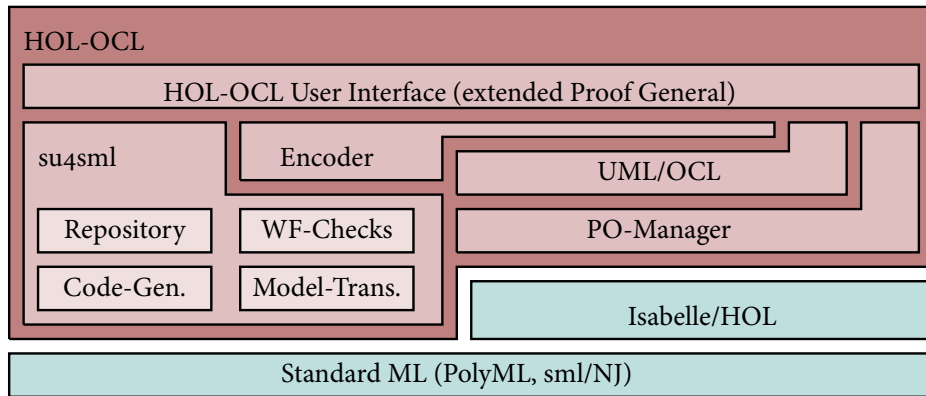

```
import_model "SimpleChair.zargo" "AbstractSimpleChair.ocl"
include_only "AbstractSimpleChair"
```
- check well-formedness and generate proof obligations for refinement:


```
analyze_consistency [data_refinement] "AbstractSimpleChair"
```
- starting a proof for a generated proof obligation:

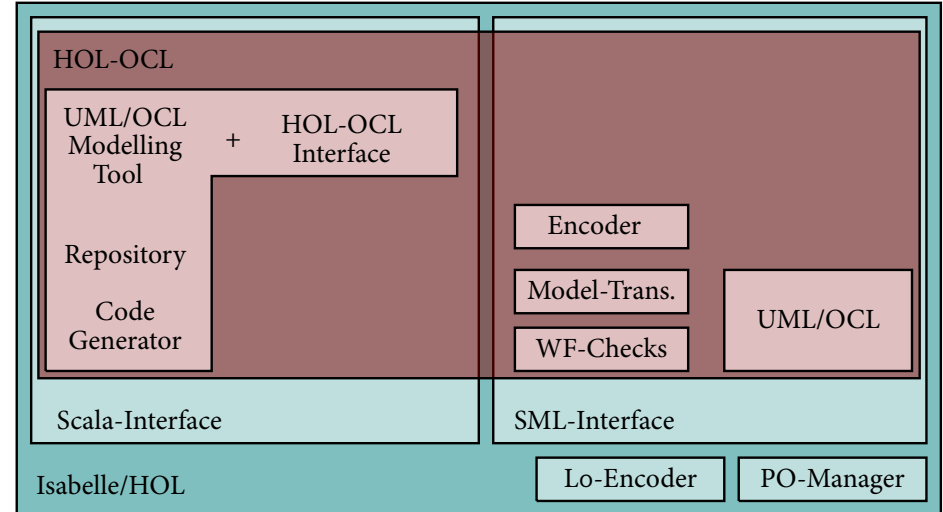

```
po "AbstractSimpleChair.findRole_enabled"
```
- generating code:


```
generate_code "java"
```

The HOL-OCL Architecture



The HOL-OCL Architecture (Next Generation)






Conclusion

- Technical challenges:
 - parsing and typing (!) concrete syntax can be slow
 - debugging simplifier setups is painful
 - defining new X-Symbol syntax is quite limited (compared to \LaTeX)
 - Best practice for communicating with external tools is missing
 - ...
- Conclusion
 - Isabelle is a framework for developing formal tools (even for tools where Isabelle not seen by the end-user)
 - The Scala-Layer enables many new features, e.g.,
 - Integration of new interaction paradigms
 - Isabelle can be (smoothly) integrated with external tools and libraries
 - ...

Thank you for your attention!

Any questions or remarks?



Bibliography I

-  Achim D. Brucker, Jürgen Doser, and Burkhard Wolff.
An MDA framework supporting OCL.
Electronic Communications of the EASST, 5, 2006.
-  Achim D. Brucker.
An Interactive Proof Environment for Object-oriented Specifications.
PhD thesis, ETH Zurich, March 2007.
ETH Dissertation No. 17097.
-  Achim D. Brucker and Burkhard Wolff.
HOL-OCL – A Formal Proof Environment for UML/OCL.
In José Fiadeiro and Paola Inverardi, editors, *Fundamental Approaches to Software Engineering (FASE08)*, number 4961 in Lecture Notes in Computer Science, pages 97–100. Springer-Verlag, 2008.

Part I

Appendix

Bibliography II

-  Achim D. Brucker and Burkhard Wolff.
An extensible encoding of object-oriented data models in HOL.
Journal of Automated Reasoning, 41:219–249, 2008.
-  Achim D. Brucker and Burkhard Wolff.
Extensible universes for object-oriented data models.
In Jan Vitek, editor, *ECOOP 2008 – Object-Oriented Programming*, number 5142 in Lecture Notes in Computer Science, pages 438–462. Springer-Verlag, 2008.

The Encoder

The model encoder is the main interface between su4sml and the Isabelle based part of HOL-OCL. The encoder

- declares HOL types for the classifiers of the model,
- encodes
 - type-casts,
 - attribute accessors, and
 - dynamic type and kind tests implicitly declared in the imported data model,
- encodes the OCL specification, i. e.,
 - class invariants
 - operation specifications
 and combines it with the core data model, and
- proves (automatically) methodology and analysis independent properties of the model.

The Library

The HOL-OCL library

- formalizes the built-in operations of UML/OCL,
- comprises over 10 000 definitions and theorems,
- build the basis for new, OCL specific, proof procedures,
- provides proof support for (formal) development methodologies.

su4sml – Overview

su4sml is a UML/OCL (and SecureUML) model repository providing

- a database for syntactic elements of UML core, namely class models and state machines as well as OCL expressions.
- support for SecureUML.
- import of UML/OCL models in different formats:
 - XMI and ArgoUML (class models and state machines)
 - OCL (plain text files)
 - USE (plain text files describing class models with OCL annotations)
- a template-based code generator (export) mechanism.
- an integrated framework for model transformations.
- a framework for checking well-formedness conditions.
- a framework for generating proof obligations.
- an interface to HOL-OCL (encoder, po manager).

Tactics (Proof Procedures)

- OCL, as logic, is quite different from HOL (e. g., three-valuedness)
- Major Isabelle proof procedures, like `simp` and `auto`, cannot handle OCL efficiently.
- HOL-OCL provides several UML/OCL specific proof procedures:
 - embedding specific tactics (e. g., unfolding a certain level)
 - a OCL specific context-rewriter
 - a OCL specific tableaux-prover
 - ...

These language specific variants increase the degree of proof for OCL.

su4sml – Code Generators

su4sml provides a template-based code generator for

- Java, supporting
 - class models and state machines
 - OCL runtime enforcement
 - SecureUML
- C#, supporting
 - class models and state machines
 - SecureUML
- USE
- ...

su4sml – Model Transformations

su4sml provides a framework for model transformation that

- supports the generation of proof obligations
- can be programmed in SML.

Currently, the following transformations are provided:

- a family of semantic preserving transformations for converting associations (e. g., n -ary into binary ones)
- a transformation from SecureUML/ComponentUML to UML/OCL.

su4sml – Proof Obligation Generator

su4sml provides an framework for proof obligation generation:

- Generates proof obligation in OCL plus minimal meta-language.
- Only minimal meta-language necessary:
 - Validity: \models , \models
 - Meta level quantifiers: \exists , \exists
 - Meta level logical connectives: \vee , \wedge , \neg
- Examples for proof obligations are:
 - (semantical) model consistency
 - Liskov's substitution principle
 - refinement conditions
 - ...
- Can be easily extended (at runtime).
- Builds, together with well-formedness checking, the basis for tool-supported methodologies.

su4sml – Well-formedness Checks

su4sml provides an framework for extended well-formedness checking:

- Checks if a given model satisfies certain syntactic constraints,
- Allows for defining dependencies between different checks
- Examples for well-formedness checks are:
 - restricting the inheritance depth
 - restringing the use of private class members
 - checking class visibilities with respect to member visibilities
 - ...
- Can be easily extended (at runtime).
- Is integrated with the generation of proof obligations.

Outline

- 5 The HOL-OCL Architecture (Details)
- 6 Mechanized Support for Model Analysis Methods
- 7 Applications of HOL-OCL

Motivation

Observation:

- UML/OCL is a *generic* modeling language:
 - usually, only a sub-set of UML is used and
 - there is no standard UML-based development process.
- Successful usage of UML usually comprises
 - a well-defined development process and
 - tools that integrate into the development process.

Conclusion:

- Formal methods for UML-based development should
 - support the local UML development methodologies and
 - integrate smoothly into the local toolchain.

A toolchain for formal methods should provide tool-support for **methodologies**.

Proof Obligations for Models

Proof Obligation Generation

- Enforce **semantical** restriction on (valid) UML/OCL models.
- Build the basis for formal development methodologies.
- Require formal tools (theorem prover, model checker, etc).

Example

- Liskov's substitution principle.
- Model consistency
- Refinement.
- ...

Well-formedness of Models

Well-formedness Checking

- Enforce **syntactical** restriction on (valid) UML/OCL models.
- Ensure a minimal quality of models.
- Can be easily supported by automated tools.

Example

- There should be at maximum five inheritance levels.
- The Specification of public operations may only refer to public class members.
- ...

Proof Obligations: Liskov's Substitution Principle

Liskov substitution principle

Let $q(x)$ be a property provable about objects x of type T . Then $q(y)$ should be true for objects y of type S where S is a subtype of T .

For constraint languages, like OCL, this boils down to:

- *pre-conditions* of overridden methods must be *weaker*.
- *post-conditions* of overridden methods must be *stronger*.

Which can formally expressed as implication:

- Weakening the pre-condition:

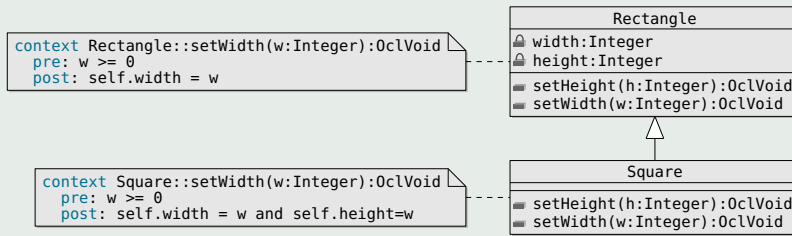
$$op_{pre} \longrightarrow op_{pre}^{sub}$$

- Strengthening the post-condition:

$$op_{post}^{sub} \longrightarrow op_{post}$$

Proof Obligations: Liskov's Substitution Principle

Example



- Weakening the pre-condition:

$$(w \geq 0) \longrightarrow (w \geq 0)$$

- Strengthening the post-condition:

$$(\text{self.width} = w \text{ and } \text{self.height} = w) \longrightarrow (\text{self.width} = w)$$

Methodology

A tool-supported methodology should

- integrate into existing toolchains and processes,
- provide a unified approach, integrating ,
 - syntactic requirements (well-formedness checks),
 - generation of proof obligations,
 - means for **verification** (proving) or **validation**, and of course
- all phases should be supported by tools.

Example

A package-based object-oriented refinement methodology.

Refinement – Motivation

Support top-down development from an abstract model to a more concrete one.

- We start with an abstract transition system

$$\text{sys}_{\text{abs}} = (\sigma_{\text{abs}}, \text{init}_{\text{abs}}, \text{op}_{\text{abs}})$$

- We refine each abstract operation op_{abs} to a more concrete one: op_{conc} .
- Resulting in a more concrete transition system

$$\text{sys}_{\text{conc}} = (\sigma_{\text{conc}}, \text{init}_{\text{conc}}, \text{op}_{\text{conc}})$$

- Such refinements can be chained:

$$\text{sys}_1 \rightsquigarrow \text{sys}_2 \rightsquigarrow \dots \rightsquigarrow \text{sys}_n$$

E.g., from an abstract model to one that supports code generation.

Refinement: Well-formedness

If package B refines a package A , then one should be able to substitute every usage of package A with package B .

- 1 For each **public class** c of A , B must provide a corresponding public class c' .
- 2 Types of **public attributes** and **public operations** (arguments and return type) must be either basic datatypes or public classes.
- 3 For **each public** class c of A , we require that the corresponding class c' of B provides at least
 - 1 public attributes with the same name and
 - 2 public operations with the same name.
- 4 The types of corresponding and attributes and operations are compatible.

Refinement: Proof Obligations – Consistency

A transition system is consistent if:

- The set of initial states is non-empty, i. e.,

$$\exists \sigma. \sigma \in \text{init}$$

- The state invariant is satisfiable, i. e.,
the conjunction of all invariants is invariant-consistent:

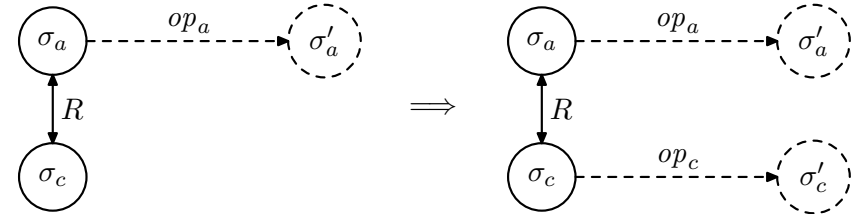
$$\exists \sigma. \sigma \models \text{inv}_1 \wedge \exists \sigma. \sigma \models \text{inv}_2 \wedge \dots \wedge \exists \sigma. \sigma \models \text{inv}_n$$

- All operations op are implementable, i. e.,
for each satisfying pre-state there exists a satisfying post-state:

$$\forall \sigma_{\text{pre}} \in \Sigma, \text{self}, i_1, \dots, i_n. \sigma_{\text{pre}} \models \text{pre}_{op} \longrightarrow \\ \exists \sigma_{\text{post}} \in \Sigma, \text{result}. (\sigma_{\text{pre}}, \sigma_{\text{post}}) \models \text{post}_{op}$$

Refinement: Proof Obligations – Implements

- Given an abstraction relation $R : \mathbb{P}(\sigma_{\text{abs}} \times \sigma_{\text{conc}})$
relating a concrete state S and an abstract states T .
- A forward refinement $S \sqsubseteq_{FS}^R T \equiv po_1(S, R, T) \wedge po_2(S, R, T)$
requires two proof obligations po_1 and po_2 .
- Preserve Implementability (po_1):**

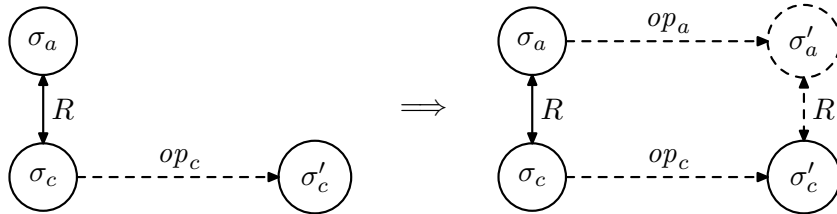


$$po_1(S, R, T) \equiv \forall \sigma_a \in \text{pre}(S), \sigma_c \in V.$$

$$(\sigma_a, \sigma_c) \in R \longrightarrow \sigma_c \in \text{pre}(T)$$

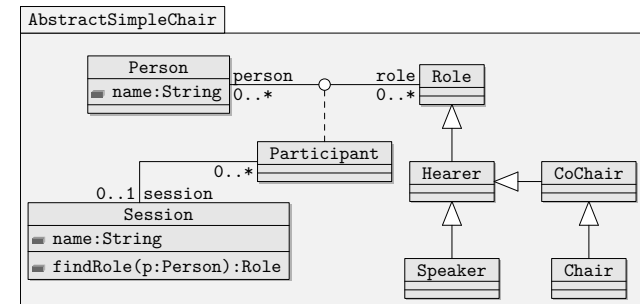
Refinement: Proof Obligations – Refines

- Given an abstraction relation $R : \mathbb{P}(\sigma_{\text{abs}} \times \sigma_{\text{conc}})$
relating a concrete state S and an abstract states T .
- A forward refinement $S \sqsubseteq_{FS}^R T \equiv po_1(S, R, T) \wedge po_2(S, R, T)$
requires two proof obligations po_1 and po_2 .
- Refinement (po_2):**



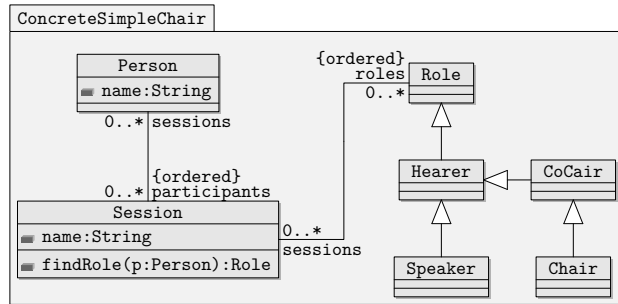
$$po_2(S, R, T) \equiv \forall \sigma_a \in \text{pre}(S), \sigma_c \in V. \sigma'_c. (\sigma_a, \sigma_c) \in R \\ \wedge (\sigma_c, \sigma'_c) \models_M T \longrightarrow \exists \sigma'_a \in V. (\sigma_a, \sigma'_a) \models_M S \wedge (\sigma'_a, \sigma'_c) \in R$$

Refinement Example: Abstract Model



```
context Session::findRole(person:Person):Role
pre: self.participates.person->includes(person)
post: result=self.participants->one(p:Participant |
      p.person = person ).role
and self.participants = self.participants@pre
and self.name = self.name@pre
```

Refinement Example: Concrete Model



```
context Session::findRole(person:Person):Role
pre: self.participants->includes(peson)
post: result ≐ roles.at(participants.indexOf(p))
```

Outline

- 5 The HOL-OCL Architecture (Details)
- 6 Mechanized Support for Model Analysis Methods
- 7 Applications of HOL-OCL

Refinement Example: Theory Sketch

```
theory SimpleChairRefinement imports OCL_methodology begin
import_model "SimpleChair.zargo" "SimpleChair.ocl"
refine "AbstractSimpleChair" "ConcreteSimpleChair"
po Refinement.findRole
```

$$\forall \sigma \in \text{pre } S, \sigma' \in \text{pre } T. R_{\text{Session}} \sigma \sigma' \text{ self self'}$$

$$\forall \sigma \in \text{pre } S, \sigma \in \text{pre } T. R_{\text{Person}} \sigma \sigma' p p'$$

$$\forall \sigma \in \text{pre } S, \sigma \in \text{pre } T. R_{\text{Role}} \sigma \sigma' \text{ result result'}$$

$$\text{AbstractSimpleChair. Session. findRole self } p \text{ result}$$

$$\sqsubseteq_{FS}^R \text{ConcreteSimpleChair. Session. findRole self' } p' \text{ result'}$$

apply(...)
discharged

Simple Consistency Analysis I

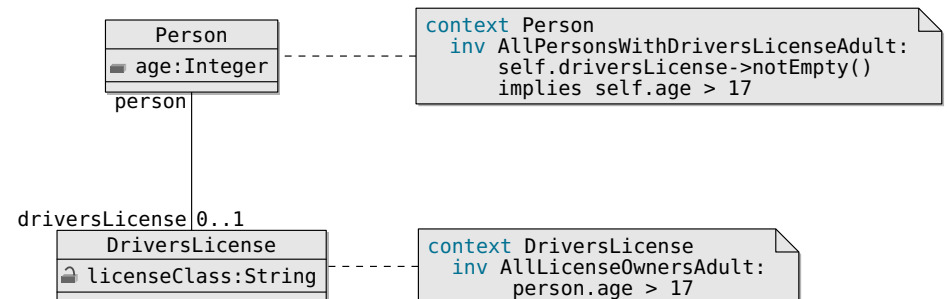


Figure: A simple model of vehicles and licenses

Simple Consistency Analysis II

```

lemma
assumes "τ = (Vehicles.Person.driversLicense(
    Vehicles.DriversLicense.person self)).IsDefined()"
    and "τ = (Vehicles.Person.age
    (Vehicles.DriversLicense.person self)).IsDefined()"
shows "τ = Person.inv.AllPersonsWithDriversLicenseAdult (
    Vehicles.DriversLicense.person self)
    → τ = DriversLicense.inv.AllLicenseOwnersAdult self"
apply(auto elim!: OclImpliesE)
apply(cut_tac prems)
apply(auto simp: inv.AllPersonsWithDriversLicenseAdult_def
    inv.AllLicenseOwnersAdult_def
    elim!: OclImpliesE SingletonSetDefined)
done

```

Liskov's Substitution Principle I

```

context A::m(p:Integer):Integer
pre: p > 0
post: result > 0

context A::m(p:Integer):Integer
pre: p >= 0
post: result = p*p + 5

-- The following constraints overrides the specification for
-- m(p:Integer):Integer that was originally defined in
-- class A, i.e., C is a subclass of A.
context C::m(p:Integer):Integer
pre: p >= 0
post: result > 1 and result = p*p+5

```

Liskov's Substitution Principle II

```

import_model "overriding.zargo" "overriding.ocl"

generate_po_liskov "pre"
generate_po_liskov "post"

po "overriding.OCL_liskov-po_lsk_pre-1"
  apply(simp add: A.m_Integer_Integer.pre1_def
    A.m_Integer_Integer.pre1.pre_o_def
    C.m_Integer_Integer.pre1_def
    C.m_Integer_Integer.pre1.pre_o_def
    A.m_Integer_Integer.pre1.pre_1_def)
  apply(ocl_auto)
discharged

```